NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN EFFECTIVE ACCESS CONTROL MECHANISM AND MULTILEVEL SECURITY FOR MULTILEVEL SECURE DATABASES

by

Matthew J. Kohler

and

Shawn W. Stroud

December 1989

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
AN EFFECTIVE ACCESS CONTROL MECHANISM AND MULTILEVEL SECURITY FOR MULTILEVEL SECURE DATABASES

12. PERSONAL AUTHOR(S)
Kohler, Matthew J. and Stroud, Shawn W.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	1989, December	81

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Database Security; Multilevel Security; Computer Security; Access Control.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Security mechanisms on contemporary database systems typically inhibit system performance. However, without security, the database system which stores classified data of different classifications, will pass through classified data of different levels in order to find the properly classified data. This 'pass through' inhibits performance (i.e., reduces access precision) because unnecessary material has been retrieved which does not aid in the resolution of the query. Further, the pass-through issue also breaches the security, since the system may pass through higher classified data for the purpose of locating the lower classified one.

This thesis shows for the first time, our Query Modification and Multilevel Security approach to database security, implemented into a single database system, i.e., the Multi-Backend, Multi-Lingual, Multi-Model Database System. These security mechanisms, unlike those seen on contemporary database systems,

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. David K. Hsiao	22b. TELEPHONE (Include Area Code) (408) 646-2253 22c. OFFICE SYMBOL Code 52Hq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-24.

UNCLASSIFIED

#19 - ABSTRACT - (CONTINUED)

enhance system performance while simultaneously providing a higher degree of security without the pass-through problem.

Approved for public release; distribution is unlimited

An Effective Access Control Mechanism and Multilevel Security for Multilevel Secure Databases

by

Matthew J. Kohler
Lieutenant, United States Navy
B.S., Indiana University of Pennsylvania, 1983

and

Shawn W. Stroud
Captain, United States Marine Corps
B.S., United States Naval Academy, 1982

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1989

Authors: _____
Matthew J. Kohler

_____
Shawn W. Stroud

Approved by: _____
David K. Hsiao, Thesis Advisor

_____
T. Wu, Second Reader

_____
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

Security mechanisms on contemporary database systems typically inhibit system performance. However, without security, the database system which stores classified data of different classifications, will pass through classified data of different levels in order to find the properly classified data. This 'pass through' inhibits performance (i.e., reduces access precision) because unnecessary material has been retrieved which does not aid in the resolution of the query. Further, the pass-through issue also breaches the security, since the system may pass through higher classified data for the purpose of locating the lower classified one.

This thesis shows for the first time, our Query Modification and Multilevel Security approach to database security, implemented into a single database system, i.e., the Multi-Backend, Multi-Lingual, Multi-Model Database System. These security mechanisms, unlike those seen on contemporary database systems, enhance system performance while simultaneously providing a higher degree of security without the pass-through problem.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. AN OVERVIEW OF THE ISSUES

This thesis articulates continued research in the area of database access control and multilevel security initiated by Hoppenstand and Hsiao [Refs. 1,2]. As addressed in [Refs. 1,2] and characterized and identified in [Ref. 3], the two outstanding technical issues in access control continue to motivate research and further study.

The first issue is **access precision.** Access precision is defined as the ratio of the amount of accessed data that satisfies a user's query versus the amount of data that has been retrieved from the secondary storage and placed in the main memory in response to the query.

The second issue is **pass-through.** When classified data with different classifications is stored in a database, it is necessary for a contemporary database system to 'pass through' other classified data in order to find the properly classified data. Anytime a database system brings classified data of different levels into the main memory from the secondary storage, the security of the system has been compromised.

These two issues greatly effect the efficiency and effectiveness of a database system. The larger the amount of data needlessly retrieved from the secondary storage in

response to a query, the smaller the access precision. If the amount of data necessary to provide a solution to a query was 'A' and the total amount of data retrieved was ('A' + 'B') where 'B' was data retrieved that made no contribution to the solution, it can be seen that the quotient {'A'/('A'+ 'B')} will yield a value less than one. **Absolute** precision, a ratio of accessed data versus data retrieved from storage equalling one, is the most desirable level of precision and is indicative of a highly efficient system. The pass-through problem not only breaches the effectiveness of system security, but also incurs less than absolute access precision.

The proposed solution to the improvement of access precisions and elimination of pass-through issues in [Refs. 1,2] was the mathematical notion of equivalence relations developed for database-system environments. An equivalence relation partitions data into mutually exclusive subsets of data. In the database environment, the equivalence relations are identified on the basis of attribute values and attribute-value ranges, thereby allowing the database to be partitioned into mutually exclusive compartments of like attribute values and value ranges. Herein lies the elegance of the solution, these attributes which make up the compartments are security attributes. The mutually exclusive compartments are collections of records where each compartment has the same aggregate of security attributes.

No two compartments have a common record with the same aggregate of security attributes. By parsing the security attributes of predicates of a user query, the database system need only access the compartments which contain the records pertaining to the query. Records not pertaining to the query will not be passed through, thus eliminating the pass-through problem. Normally, in conventional database systems there has to be trade-offs between performance and security. One way that systems try to increase performance is by caching data retrieved by one query in hope that the next query will probably pertain to some of the information already retrieved in the prior query thus saving on the retrieval time. Here security is definitely sacrificed on behalf of performance. It is believed that high-performance database systems do not support secure databases and vice-versa. Hoppenstand's research discovered that the concept of equivalence relations and secure compartments did not sacrifice performance but increased it since the total amount of data needing to be retrieved by a query was decreased. Thus the myth that security will penalize performance was destroyed.

B. MOTIVATION FOR MULTILEVEL SECURITY

The need of **multilevel security** database systems is due to the military's security classifications. The military normally classifies its information into one of four classes. These classes are unclassified (UC), confidential

(C), secret (S), and top-secret (TS). These classes (also called levels) are of increasing degrees of sensitivity and require increasing authority to clear them for accesses. Also an individual may have a certain clearance level and still not have access to certain information classified at that level. This is known as the **need-to-know** requirement. It provides the motivation for **query modification** which will be discussed in Chapter III. A multilevel-secure system is defined by [Ref. 4] as "a system containing information with different sensitivities that simultaneously permits access by users with different security clearances and needs-to-know, but prevents users from obtaining access to information for which they lack authorization." Since this is exactly the thought behind DoD's security classification system (UC,C,S,TS), a database that incorporates this concept would be appropriate for military applications with efficiency and effectiveness.

Hoppenstand expanded his security attribute into a multilevel structure and showed how clustering of records into secure compartments in each classification level would, in theory, allow an implementation of a multilevel-security system.

C. REASONS FOR ACCESS CONTROL

The multilevel-security database discussed above was concerned with the structure of a database that supports the secure storage and retrieval of data with different

classification levels without breaching the security by passing-through the different levels during retrieval and without slowing the performance of the system. The security clearance of the user has not been discussed at this point. Nor has the secure access operations and controls of the database system been discussed at this point.

There is a need for an access control mechanism for the database system to regulate and verify accesses to the multilevel secure database. A user may not be allowed access to certain information within a classification for which he has clearance. The following is an example: User 1 has a top-secret (TS) clearance for information pertaining to military aircraft. He does not, however, have TS clearance for information on the new B2 bomber which is also classified TS. If a system only verifies a user's security clearance without upholding the need-to-know requirement, the system would probably release TS information about the B2, if queried, to User 1 because he does have TS clearance. In order for a database system to be efficient and effective with respect to security, it must regulate and verify accesses for each user of the system, i.e., there must be **access controls** of finer granules (units) of the database (information) within the system. Access controls are involved in the assignment of users' rights and privileges for the purpose of controlling access to the database on the

basis of the user's need-to-know requirement and multilevel security clearance.

After review of current access control mechanisms, the Query Modification [Refs. 5,6] was chosen. In the query modification mechanism, a user's query is modified according to user's data access rights by appending a permit clause as a conjunction to the user's query. The 'modified' query is then submitted to the query processor of the system as the query to be executed. Mathematically, the system response is the intersection of the information determined by the original query and the information determined by the permit clause.

The query modification mechanism is implemented in an experimental database management system which is discussed in the following sections.

D. CHOOSING A SYSTEM FOR EXPERIMENTING THE MULTILEVEL SECURE DATABASES AND ITS ACCESS CONTROL MECHANISM

In order to demonstrate the notion of security attributes, Hoppenstand chose D.K. Hsiao's Attribute-Based Data Model [Ref. 7] (ABDM) for its characterization. There were two reasons why ABDM was chosen. The first reason, which is still applicable to this thesis, is that ABDM is at the cutting edge of database technology. There is an experimental system that has moved pass the present conventional systems in theory and design, which employs ABDM databases. In database research, it is more desirable

to make developments and improvements to a system that will be involved in the future of database technology, not a system that is to be replaced by newer technology.

The second reason for the selection was that the security attributes and equivalence classes required for Hoppenstand's solution to the access control and pass-through problem could be easily defined in ABDM's meta data. The meta data is stored information which describes the base data of the database. It consists primarily of a group of tables that is maintained by the database system to govern record clustering and to store record identifiers (e.g., addresses). The meta-data tables form the directory of the database and these tables are made up of attributes, descriptors, and cluster addresses. These clusters are groups of records such that every record in a cluster satisfies the same set of descriptors. Therefore, these clusters constitute the base data. This experimental database system supports exactly Hoppenstand's idea of security attribute and secure compartmentalization and was therefore our system of choice.

E. THE SYSTEM ARCHITECTURE

Since ABDM was selected as the test data model for the multilevel secure databases, further explanation is needed in the area of its system architecture. The current experimental system being used is referred to as the Multi-Lingual, Multi-Model, Multi-Backend system and it uses

the Attribute-Based Data Language (ABDL) as its kernel language [Ref. 8]. The system supports multiple database languages and models, and is configured to run with multiple **backends.** It is not the intention of this thesis to address the multi-lingual or multi-model aspects of this system, since they are not relevant to the thesis work. It is, however the intention to examine the idea of the backend concept which is relevant to our thesis work. The layout of the system can be seen in Figure 1.1. Each database system consists of zero or one (backend) controller and one or more backends. The backend controller and the backends are interconnected by an ethernet interconnection.

The controller serves as a communications frontend (therefore, often referred to as the 'frontend') for the backends as well as serving the controlling function of the database. It normally has a tape unit which is used for bootstrapping the controller and backend software. The tape unit can also be used for backing up the system in protection against system malfunction. The backends are the "database engines" of the multiple-backend database system [Ref. 8]. They are microprocessor-based workstations with large amounts of storage capacity. The backend computers are dedicated to the tasks of data storage and access. Each backend has three disk drives, one for its evenly distributed portion of the clustered base data, another for the meta data which describes the base data of the database

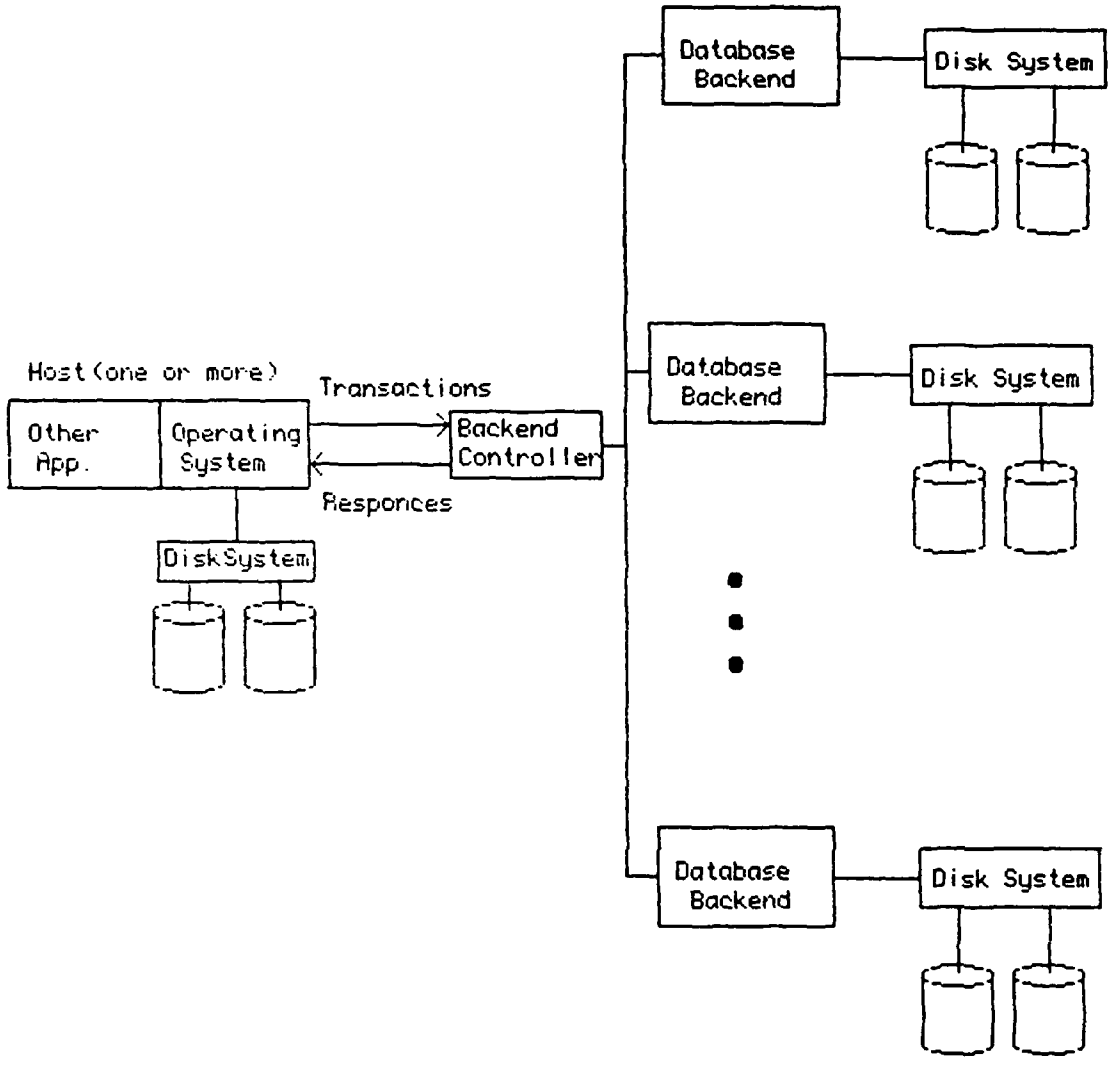

Figure 1.1 Multi-Backend System Layout

and is replicated on every backend, and the third is for the operating system to do paging. The system will work with one backend; however, additional backends and their disks improve the performance of the system. Since base data (different clusters of files of records) is spread evenly over the number of backends in existence, a search for and retrieval of data for a query is done in parallel on all the backends at once. This decreases the overall retrieval time of the solution and thus increases performance. Actual performance evaluations of the multi-backend system can be found in [Ref. 9].

F. THE METHODOLOGY USED

This thesis implements the proposal of Hoppenstand on the structures of multilevel security into Hsiao's experimental Multi-Lingual, Multi-Model, Multi-Backend system. It also incorporates a design for an effective and efficient access control mechanism into the frontend of the experimental database.

This will be the first time that a frontend (Query Mod) and backend (multilevel security attributes) approach to database security has been implemented into a single database system. We believe this system will now demonstrate not only high performance with multilevel security but also more effectiveness within access control and security compartmentalization. In other words, efficiency and effectiveness can be achieved in a multilevel

database system if our compartmentalization technique and query modification method are employed in database organization and access control, respectively.

G. SUPPORTING WORK

Characterization and identification of the pass-through and access precision problems is due to D.K. Hsiao [Ref. 3] who also outlined the security atom concept which was the fundamental basis of the security methodology outlined in Hoppenstand's paper [Ref. 1] and therefore continued throughout this thesis.

H. ACKNOWLEDGMENTS

The authors would like to acknowledge and express their sincere thanks to Thomas Chu for his technical support on the Multi-Backend, Multi-Lingual, Multi-Model Database System of the Laboratory for Database Systems Research at the Naval Postgraduate School and to Debbie Gaiser and Marciano Pitargue for their support in the preparation of this thesis and for their continued dedication to the project even after leaving the staff of the Laboratory for Database Systems Research.

II. GENERAL DESCRIPTION OF ABDM & MBDS

As discussed in the Introduction of this thesis, the Attribute-Based Data Model (ABDM) was chosen to demonstrate the notion of security attributes and to implement the access control mechanism. In order to better understand ABDM structure and support further discussion in later chapters of this thesis, this model is now discussed in detail.

A. THE BASE DATA

ABDM characterizes two kinds of data: the **base data** and the **meta data**. The base data is what is normally thought of when discussing the data within a traditional database. Basically, it is information that is stored, in order to be retrieved and used at a later time. Specifically in ABDM, it is a collection of files of records. Each record is composed of a collection of attribute-value pairs and textual information. The attribute-value pair is sometimes referred to as a keyword and the textual information referred to as the record body.

The attribute-value pair is a member of the Cartesian product of the attribute set and the value domain of the attribute. A record contains at most one attribute-value pair for each attribute defined in the database, i.e., no

two attributes are identical within an individual record. An example of a record is shown below.

(<TEMP,AircraftInfo>,<RADIUS,1200>,<PLANE,Fighter>,
<COUNTRY,USA>, (Carrier Ops Certified))

The angle brackets, <,>, enclose an attribute-value pair. The curly brackets, {,}, include the record body. The entire record is enclosed in parentheses, (,). It should be noted that the first attribute-value pair of all records of a file is the same. The attribute is TEMP and its value is the file name. The above example therefore has the file name of AircraftInfo.

All constructs of the ABDM relating to the base data are well defined in terms of other constructs with two exceptions. The two undefined constructs are the attribute set and the value domain.

There are certain attribute-value pairs of a record that are called the **directory keywords** of the record. They are referred to by this name because either the attribute-value pairs or their attribute-value ranges are kept in a **directory** for identifying the records. Consequently, those attribute-value pairs which are not kept in the directory are called **non-directory keywords.** The records of the files of a database constitute the base data of the database. In general, given a record of attribute-value pairs, one cannot

keywords unless one refers to the meta data which is to be discussed in the next section.

B. THE META DATA

The **meta data** is defined as stored information about the base data. It is the various meta-data constructs that form the **directory** of the database, referred to in the previous section. The constructs of the directory are: attributes, descriptors, and clusters. Collectively, they contain all information needed to support the database interaction with the base data.

An **attribute** is used to represent a category or certain common property of the base data. Using the AircraftInfo file example from the previous section, RADIUS is an attribute that corresponds to actual aircraft radius features in the database. A **descriptor** is used to describe a range of values or a unique value that an attribute may have. For example (1001 <= RADIUS <= 1200) is a possible descriptor for the attribute RADIUS. The descriptors that are defined for an attribute are mutually exclusive in terms of their values, i.e., you would not see the following two descriptors within the same meta data:

(1001 <= RADIUS <= 1200) and (750 <= RADIUS <= 1100).

The reason is, an aircraft with a radius of 1050 for example, would fall into both descriptors ofthe attribute RADIUS and therefore mutual exclusivity would not hold. It should be noted that there are no restrictions as to the

should be noted that there are no restrictions as to the number of descriptors for an attribute within a database only that the descriptors be mutually exclusive in terms of their values. Mathematically, the descriptors of the attribute serve to derive equivalence classes which effectively partition the database into mutually exclusive sets of records. These record sets are referred to as clusters and will be discussed next.

Probably the single most important construct in the meta data is the cluster. Although the other constructs are important in that they allow the notion of a cluster to be defined, it is the concept of the cluster that the entire system is built around. It will be seen in follow-on chapters how the cluster supports the design and implementation of multi-level security and the access control mechanism. A **cluster** is a group of records such that every record in the cluster satisfies the same set of descriptors. Looking at the attribute RADIUS again, all records with RADIUS between 1001 and 1200 may form one cluster whose descriptor is (1001 <= RADIUS <= 1200). In this case, the cluster satisfies the set of a single descriptor. In a typical ABDM database, a cluster tends to satisfy a set of multiple descriptors.

An important characteristic of the cluster is the non-uniformity of its size. It would not be logical to pre-define the size of a cluster, in that its size is

dynamic with respect to the amount of data within the database that satisfies the descriptor set of that cluster. Some clusters may have hundreds or thousands of records; other clusters may have only a few; while still others may not have any records at all.

ABDM requires some means of effectively keeping track of the meta data constructs. To fulfill this requirement, three tables are maintained and it is these tables that form the directory mentioned in the beginning of this section. The three tables, which define a database, are: **the attribute table** (AT), **the descriptor-to-descriptor-id table** (DDIT) and **the cluster-definition table** (CDT), examples of which can be seen in Figure 2.1.

The attribute table is used to keep track of the attributes for which descriptors are formed; more specifically, the attribute table maps directory attributes to the descriptors defined on them. Observe Figure 2.1.a, AT here contains only three attributes (i.e., Radius, Plane, Temp), this is for illustration purposes only, the typical database would have considerably more attributes.

The descriptor-to-descriptor-id table is used to keep track of all the descriptor sets that effectively partition the database into clusters; more specifically, DDIT maps each descriptor to a unique descriptor id. There are three types of descriptors available within DDIT. They are referred to as: Type-A, Type-B, and Type-C descriptors. A

Attribute	Attribute Type	DDIT Entry
RADIUS	A	D11
PLANE	C	D21
TEMP	B	D31

(a) An Attribute Table (AT).

Id	Descriptor
D11	0≤RADIUS≤ 600
D12	601≤RADIUS≤1000
D13	1001≤RADIUS≤1200
D14	1201≤RADIUS
D21	PLANE = Fighter
D22	PLANE = Bomber
D23	PLANE = Recon
D31	TEMP = U S A
D32	TEMP = U S S R

(b) A Descriptor-to-Descriptor-Id Table (DDIT).

Id	Desc-Id Set	Record-Id
C1	{D11,D21,D31}	R1,R2,R3,R4
C2	{D11,D21,D32}	R5,R6
C3	{D12,D21,D31}	R7,R8,R9
C4	{D12,D21,D32}	R10,R11
C5	{D13,D22,D31}	R12,R13,R14
C6	{D13,D22,D32}	R15,R16
C7	{D14,D21,D31}	R17
C8	{D14,D21,D32}	R18,R19,R20,R21,R22
C9	{D13,D23,D31}	R23
C10	{D13,D23,D32}	R24,R25
C11	{D14,D23,D31}	R26
C12	{D14,D23,D32}	R27,R28

(c) A Cluster-Definition Table (CDT).

Figure 2.1 The Directory Tables

type-A descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate. The following is an example of a type-A descriptor: ((RADIUS > 1001) and (RADIUS < 1200)). Note that the same attribute (in this case RADIUS) must appear in both predicates. A type-B descriptor consists of an equality predicate. An example of this is (TEMP = USA). The type-C descriptor consists of the name of an attribute only. The type-C attribute defines a set of type-C sub-descriptors. These are equality predicates defined over all unique attribute values which exist in the database. This then allows by definition the type-C sub-descriptor to also be of type-B. Although a type-C descriptor is also of type-B, unlike type-B, type-C descriptors are on-demand descriptors, i.e., if for a type-C attribute, a type-C sub-descriptor for a particular value (this new value could have either been inserted by the user during an update operation or in a record generation session) is not available in DDIT, a new type-C sub-descriptor is automatically generated and inserted into DDIT. An example of the type-C sub-descriptor is as follows: the type-C attribute PLANE forms the type-C sub-descriptors (PLANE = Fighter), (PLANE = Bomber), (PLANE = Recon) where "Fighter," "Bomber" and "Recon" are the only unique database value of the PLANE attribute in the base data. Finally, observe that as additional descriptors (no matter what the type) for the same attribute are introduced,

one of the partitions within DDIT is further divided into two. It can now be observed that for a given attribute, the number of resulting partitions is proportional to the number of descriptors introduced.

The cluster-definition table is used to keep track of which records belong to which clusters; more specifically, CDT maps descriptor-id sets to cluster ids and cluster ids to record ids. An example of CDT can be seen in Figure 2.1.c. Each entry in CDT consists of a unique cluster id, e.g., C1, C2, C3, ..., a set of descriptor ids whose descriptors define the cluster, and ids of the records that are in the cluster. The record ids within the cluster will actually correspond to the address locations where the records will reside on the base data disk. CDT only keeps track of those clusters that contain records. If new records are entered into the database that require clusters not currently defined within CDT, then at that time new clusters are formed and entered into CDT. As the number of descriptors which characterizes the clusters increases, the sizes of the clusters decrease. Individually, each of them is smaller than a typical database. Within a database, it can therefore be seen why a more general descriptor is less desirable than a specific one. There will be less records to retrieve for a particular Descriptor-Id set since on the average it should be smaller. For example referring to Figure 2.1.b, if there was a single descriptor (0 <= RADIUS

<= 1000) instead of D11 and D12, there would be less clusters and more records within the remaining clusters. For example C1 and C2 would merge into one cluster with seven records instead of two clusters with four and three records, respectively.

C. THE ATTRIBUTE-BASED DATA LANGUAGE (ABDL)

When examining the Attribute-Based Data Model (ABDM), its native (kernal) language, the attribute-based data language (ABDL), should also be included in the examination. ABDL supports the system's five primary database operations: INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. Individually, each operation is considered a **request** Collective, two or more requests together form a **transaction.** Except for the actual database creation and control operations which are carried out by menu-driven language not in the same style of the rest of the data language, these five operations can provide the user with all the necessary requests and transactions for the manipulation of a typical database. It should be noted here that the database creation as well as the exercising of access control over the database is a task of the database administrator. This idea will be expanded upon in Chapter III, i.e., the access control chapter.

In order to promote a "feel" for the language, not a detailed understanding of its formal specifications, each of the operations will be illustrated with a simple example.

The INSERT request is used to insert a new record into an existing database. Each request in ABDL has along with an operation a qualification. This qualification is used to specify the part of the database that is to be operated upon. Within the INSERT operation, the qualification is a list of attribute-value pairs and a record body being inserted. The following is an example of an INSERT operation:

```
INSERT (<TEMP,AircraftInfo>,<RADIUS,1200>,<PLANE,Fighter>,
        <Country,USA>, (Carrier Ops Certified))
```

The operation will insert a record into the AircraftInfo database file that has the characteristics of it being a fighter aircraft of the US with the radius of 1200. Note the fact that this aircraft is carrier-ops-certified has no baring on the record's characteristics since this information is in the record body and not found in the attribute-value pairs.

An UPDATE request is used to modify records of a database. Its qualification consists of the query and the modifier. The query specifies which records of the database are to be modified, the modifier specifies how the records being modified are to be updated. The following is an example of the UPDATE operation:

```
UPDATE (TEMP = AircraftInfo) (PLANE = Fighter) (COUNTRY =
                USA) (RADIUS = RADIUS + 500)
```

This operation will increase each US fighter aircraft's radius by 500. Here the query is (TEMP = AircraftInfo) and the modifier is (RADIUS = RADIUS + 500).

The RETRIEVE operation does as its name implies, it is used to retrieve records of a database. The qualification of the RETRIEVE request includes a query, and an optional target-list and by-clause. The query specifies which records are to be retrieved, the target-list consists of a list of attributes whose attributes values the user desires to be output and the by-clause may be used to group records. The following is an example of the RETRIEVE operation without the optional by-clause:

```
RETRIEVE ((TEMP = AircraftInfo) and (RADIUS > 1500))
                (PLANE)
```

This operation will produce to the user the plane names of all the records in the AircraftInfo database file with a radius of greater than 1500. Here the (TEMP = AircraftInfo) and (RADIUS > 1500) form the query and (PLANE) is the target list.

The DELETE operation is used to remove one or more records from a database. Its qualification is a query. The following is an example of the DELETE operation:

```
DELETE ((TEMP = AircraftInfo) and (RADIUS > 1500))
```

This operation similar to the RETRIEVE format, will delete from the AircraftInfo database file all records whose aircraft have a radius of greater than 1500. Here, the query is simply (TEMP = AircraftInfo) and (RADIUS > 1500).

The last of the five operations is perhaps the most complicated, the RETRIEVE-COMMON request is used to merge two files by common attribute values. It is a transaction of two retrieve operations with a common clause in between. The following is an example of the RETRIEVE-COMMON operation.

```
RETRIEVE ((TEMP = NavalAircraft) and (RADIUS > 1500))
     (PLANE)
COMMON (RADIUS,RADIUS)
RETRIEVE ((TEMP = MarineAircraft) and
     (RADIUS > 1500)) (PLANE)
```

This operation will find all records in the naval aircraft database file with aircraft radius greater than 1500, find all records in the marine aircraft database file with

aircraft radius greater than 1500, identify records from respective files with radii in common, and return the records of planes with the same radius figures. It should be noted that the common attributes and the target lists need not have identical names, although they are identical in this example.

As a final thought with respect to ABDL, this language supports other data languages which are translated into it. This helps provide the multi-lingual portion of the Multi-Backend, Multi-Model, Multi-Lingual Database System [Ref. 8]. As mentioned previously, ABDM and ABDL will not be further discussed in our thesis, although their introduction should be helpful in understanding the overall system composition.

D. AN OVERVIEW OF THE MULTI-BACKEND DATABASE SYSTEM

The Multi-backend database system (MBDS) is the hardware which supports ABDM discussed in the previous section. That is, the attribute-based data language (ABDL) of ABDM is the kernel language used in MBDS. As mentioned in the thesis introduction, MBDS consists of one or more backends and their disk subsystems which are configured in a parallel fashion and are regulated by a controller computer. It is important to note that in this hardware configuration, MBDS separates the controller hardware from the backend hardware. This allows the system not only to be beneficial for

pedagogical purposes but supports database growth and performance improvements.

1. The Backends

Parallel architecture is a key feature of MBDS and is what is instrumental in the performance improvement of the system. **Multi-backends** means several backends used at the same time, i.e., in parallel. They are dedicated to the tasks of data storage and access. Each backend maintains a copy of the meta data and the distinct clustered data assigned to it. In general, since there is a finite amount of data within a database at the time of any transaction, the complete search of the database would take X amount of time. Suppose the data was distributed evenly across several database systems and searched at the same time, i.e., each system search its portion of data at the same time. Since each system would contain less data than the original database system did, it is obvious to see that it would take less time to search the entire amount of data. This is the idea behind the multiple-backend system. Instead of individual systems containing portions of the entire base data, one system employing several backends is used. The records of a given cluster are evenly distributed over backends so that the search time of all backends running in parallel is proportionally less than one backend searching all the records of the cluster by itself. The

controller computer coordinates the communication of transactions and will be discussed next.

2. The Frontend

The controller computer does not contain any of the clustered data itself. Its function is to control the database system and communicate with the backends and the users. It should be noted that the access control mechanism, to be discussed in Chapter III, is also located in the frontend. By using a broadcast bus (presently an ethernet with special modification), the controller is able to simultaneously send a message to all the each backends. The message can be typical database transactions to be executed by the backends. The backends can then execute the transactions in a parallel fashion. It is interesting to note that if multiple transactions are sent across the bus to each backend by the controller, the backends have the ability to queue the transactions and handle them in order without re-transmission by the controller. The term **communications frontend** or just **frontend** is used to refer to the controller because of this type of communications coordination.

E. FRONTEND VS. BACKEND

The previous sections of this chapter discussed the Multi-Backend database system architecture. This explanation provided the background required to understand the design alternatives discussed in this section.

Specifically, the terms discussed previously to be considered in this section are **frontend** and **backend.**

The access control and multi-level security design decisions with respect to system architecture considered by this thesis is primarily that of frontend-versus-backend. In other words, should the access control and multi-level security mechanisms reside in the frontend, or backend? Intuitively, based on the parallelism provided by the backends, it would seem most advantageous in terms of efficiency, to place as much work as possible in the backends. This would result in the processing being distributed over the backends thus achieving the maximum benefit from the parallelism. The result of placing too many requirements in the frontend would be creating the potential of a bottleneck in the system. A bottleneck in the frontend would result in the backends being idle while the frontend was busy. Obviously, this situation should be avoided since it reduces the efficiency gains produced from the parallelism of the system.

The previous paragraph presents a pretty strong argument for placing access control and multi-level security mechanisms in the backends. However, as will be shown in the remainder of this chapter, the functionality of the mechanisms make it worth while considering both the frontend and backend when making design decisions.

We now will discuss the design decision concerning where the access control mechanism is to reside in the Multi-Model, Multi-Lingual, Multi-Backend Database System. Various alternatives will be discussed that consider both frontend and backend possibilities. Chapter III will discuss the actual design and implementation of the access control mechanism itself.

Access control, as discussed in Section A of this chapter, involves the assignment of user privileges. Therefore access control can be considered in two parts: the storage of the user privileges, and the algorithm to utilize and implement these user privileges. Though both of these parts will reside in either the frontend or backend, both need to be incorporated in the decision process of frontend versus backend.

The first issue to be discussed is the storage of user privileges. If the backend method is to be considered, then this user privilege data will have to reside in each of the backends meta data disks. Hsiao and Menon suggest several design alternatives in this regard. One of these designs will be illustrated to provide a general understanding of how this would be accomplished, and to point out the benefits and drawbacks to such a design.

The example provided here is typical of a backend design in that it requires modification to the meta data structures. i.e., AT, DDIT, and CDT. Figure 2.2 uses the

same meta data tables introduced in Section A, an shows these modifications made to DDIT and CDT.

Id	Descriptor	USER-1	USER-2
D11	0≤RADIUS≤ 600	--	no insert
D12	601≤RADIUS≤1000	--	no insert
D13	1001≤RADIUS≤1200	--	no update, no delete
D14	1201≤RADIUS	--	all
D21	PLANE = Fighter	--	no update, no delete
D22	PLANE = Bomber	--	all
D23	PLANE = Recon	--	all
D31	TEMP = U S A	--	--
D32	TEMP = U S S R	--	all

Figure 2.2 A Descriptor-to-Descriptor-Id Table (DDIT)

Here, DDIT is augmented with two columns. Each column articulates the access privileges for one user. Using the notation suggested by Hsiao and Menon, a '-' indicates that no operation is disallowed for that descriptor for that particular user. An 'ALL' indicates just the opposite, all operations are allowed for that descriptor for that user. CDT formed from this augmented DDIT can come in two forms, a single large CDT, with privilege columns for each user, as shown in Figure 2.1.c , or a separate CDT formed for each user that specifies only those clusters authorized for that user along with that user's privileges for that cluster, as shown in Figure 2.3.

Id	Desc-Id Set	Record-Id	
C1	{D11,D21,D31}	R1,R2,R3,R4	no update, no delete, no insert
C3	{D12,D21,D31}	R7,R8,R9	no insert, no update, no delete

Figure 2.3 A Cluster-Definition Table (CDT)

It is not the purpose of this section to present a full understanding of how this particular design functions. What should be noted from this example is these meta data tables function as they did prior to modification. That is, they continue to provide the absolute access precision, but now also articulate access control. This design provides a complete method for access control. Furthermore, because it incorporates these mechanisms in the backends, it makes use of the systems parallel architecture. However, this design has several other aspects which must be considered. This design exists at the expense of the secondary storage. This in and of itself is not a criterion for ruling out this design because the secondary storage is inexpensive in terms of cost and can easily be expanded. However, especially when considering a separate CDT for each user, if the number of users and the amount of data is high, this method maybecome prohibitive. This potential problem is compounded by the fact that the multi-level security design and implementation, discussed in Chapter IV, uses separate CDT's for each security level, the amount of meta data might

become excessively large. Another potential drawback to this design occurs when considering updates to a user's privilege. Updating a user's privilege involves the restructuring of the meta data tables (DDIT and CDT) for that user. This could be expensive if the update requires extensive restructuring, and during this restructuring process no other queries can be processed. This problem is exacerbated by the fact that the meta data exists at each backend.

These problems with the backend approach to access controls leads us to look for an alternative solution, which is the frontend approach. Chapter III discusses the design and implementation of the frontend approach for access control, but its immediate advantages over the backend approach will be discussed here.

First of all, it must be pointed out that the frontend/controller handles the query before it is broadcasted to the backends. The frontend method for access control modifies the query before this broadcasting occurs. This process is called Query Mod, and is described in detail in Chapter III. This frontend method has the immediate advantage of determining the validity of the query, i.e., if the query is requesting data not authorized for that user. The savings of the frontend over the backend method is that the latter will not determine query validity until after the query has been broadcasted, and processed at each backend.

Obviously, it would be beneficial to determine query validity at the frontend. This requires that the user privilege data and the access control algorithm be located in the frontend. The advantage of placing the user privilege in the frontend is that modification to this data is effected more efficiently than in the case of the backend. This is because this data is centrally maintained and therefore more easily updated. The drawback to this method is that it is not able to take advantage of the parallelism offered by the system. However, provided that the Query Mod method is efficient, the gains achieved via the advantages mentioned above, would be expected to more than outweigh this disadvantage.

III. ACCESS CONTROL

Access control in the context used in this thesis refers to controlling what accesses a user can effect once that user has gained access to the system. Therefore, it is assumed that adequate operating system security measures are in effect to prohibit unauthorized entry into the system. Separate access control mechanisms for the database system will handle an authorized entry and utilization of the database. It is the security mechanisms for the database that are discussed here.

As stated in Section D.2 of Chapter II, the access control mechanism discussed in this chapter resides in the frontend. Access control will be separated into two distinct but related areas: query modification and user profile specification. User profile specification is discussed first.

A. THE USER PROFILE SPECIFICATION

1. The Need of the Database Administrator (DBA)

The purpose of user profile specification is to allow the access privileges of a particular user to be specified, stored and maintained. Obviously these access privileges need to be monitored and dictated by some trusted, benevolent authority. This authority will be called the Database Administrator or simply DBA. It will be

DBA's responsibility to ensure that every user has a profile. A user's profile will be the statement of access privileges for that user. DBA alone (excluding trusted system processes) must have access to each user's profile for the purpose of modification of that user's access privileges.

Each user profile will need to contain specific information that will be utilized by the query modification routine which will be discussed later.

First, each user will need to have a user profile. If a person tries to logon to the database system and no user profile exists for that user, the entry into the system will not be granted. Furthermore, a password must be supplied by the user for authentication purposes. Therefore, both the user name and that user's corresponding password will need to be maintained in the user profile.

2. Two Approaches to the Specifications

Following the specification of user name and password, the next logical step is to specify which databases a given user has access to. There are two possible approaches to this issue. The first method would be to specify those databases to which the user is not allowed any access. The second method is to specify only those databases to which the user is allowed some access. Both methods would properly maintain access control based on a need-to-know criterion; however, the second method,

stating which databases the user is allowed access, is the method of choice. This is because the number of databases to which a user is allowed any access is likely to be small compared to the total number of databases in the system. This would be particularly true in a system that contains a large number of databases. The benefit of selecting this method also becomes apparent in its utilization. Because of this method, an access to a database must be explicitly stated in a user profile. When a user requests to access a specific database, an exact match based on that database name must be found in that user's profile. If that match is not found, no access will be granted. When an access to a database is requested, only half of the list in the user profile containing valid databases would have to be searched on the average. If the other method were used, stating only those databases to which no access is allowed, would require searching the entire list every time an access to a database was requested. Not only is this method less efficient, it is also less secure because it does not require an exact match for it to succeed. For example, an access to a misspelled database name would succeed at this initial step, therefore, requiring more error checking. For these reasons, it is decided to state in the user profile only those databases to which the user is allowed some access.

For access control to be realistic and complete, it must be able to specify different access privileges for each

database to which has been allowed an access. Therefore, the user profile must be able to specify access privileges for each database. These access privileges are discussed in the following paragraphs.

3. Specifying Operations and Data for Individual Databases

Access privileges on a given database are based on what operations that can be performed on that database. All database operations can be placed in four basic categories: Retrieve, Insert, Update, and Delete. Therefore, a user profile will need to specify database access privileges based on these operations. This means that it must be specified in the user profile if that user can perform some, any, or all of these operations.

However, this matter is slightly more complicated when more than one but less than all operations are to be authorized. This concerns the inferences that can be made from the allowed operations. For example, the retrieve, insert and delete operations have been authorized and the update operation has been not been authorized. In this case, the fact that the update operation has not been authorized can be subverted simply by a combination of deletions and insertions. Therefore, an algorithm must be implemented to assist DBA to safeguard against such unwanted inferences when a user profile is created or modified.

4. Specifying Finer Classifications and Granules of Data for Protections

This method of limiting operations available to the user can be extended to further articulate the access privileges of a user. Once the use of a particular database operation (retrieve, insert, etc.) has been granted, it must also be possible to restrict an access to data based on their attribute names or attribute values, or both. This would include restricting a user's access to data based on the security classification of the data requested (unclassified, confidential, secret, top secret), and/or data value ranges, i.e., restricting a user accessing data within certain value ranges. The user profile would need to specify access privileges using these methods for each attribute of each database specified in a user's profile.

5. The Detailed Design of User-Profile Specifications

The need and use of a user profile has been discussed in the high-level design discussion of the previous sections. It has been also stated what the user profile should contain. This section discusses the detailed design of the user profile. This design discusses the data structures that contain the user profile, and also the operations that can be performed on the user profile by DBA.

The user profile is stored in a file format. When a user initiates a session with the database, that the user's profile is read from the file format into a linked list

structure. The file format is introduced in Figure 3.1 to aid in explanation.

As can be seen in this figure, all the information necessary to specify a user access as required by the high-level design has been supported.

When a potential user requests an entry into the database system, that user's corresponding user profile is consulted for authentication purposes. If it is a valid user, that user profile is read into a linked list structure. This linked list structure is maintained in the memory until the user exits the system. This user-profile linked list is utilized by the query modification mechanism to be discussed later. The linked list structure is illustrated in Figure 3.2 and Figure 3.3.

As stated earlier, DBA has the responsibility and authority to perform operations on user profiles. These operations involve creating, modifying, and deleting user profiles. These operations are effected by DBA's own user profile utilizing the same linked-list data structures described in Figure 3.3. These data structures are used to assist in making corrections and modifications easier and more efficient.

When creating a new user profile, the memory is created for the user profile linked list. (See Figure 3.3 again.) DBA is queried for input, and these values are placed in the data structure. DBA is then given the

```
File: username.usrprofl
username                    /* user name                                  */
9999                        /* user identification                        */
pswd                        /* user password                              */
Databasename 1              /*name of data base user allowed to access    */
Templatename 1              /* name of a template in Databasename 1       */
retrieve y                  /* what operations user can perform           */
insert                      /* on Templatename 1. 'Y'or'y' specifies      */
update  y                   /* operation is allowed, else not allowed     */
delete
SECURITY                    /*states security level for each operation    */
retrieve 0-3                /*security ranges:  0 - unclassified         */
insert  0-3                 /*                  1 - confidential          */
update  0-3                 /*                  2 - secret                */
delete  0-3                 /*                  3 - top secret            */
attributename 1             /*a  attribute in Templatename 1             */
retrieve                    /*format operator,relational operator,range  */
insert                      /*specifies allowable data ranges for that   */
update                      /* operation                                 */
delete attributename 1 <= 9999
attributename2              /*another  attribute in Templatename 1       */
retrieve attributename2 < L
insert
update
delete
@                           /* end of Templatename 1                     */
Templatename2               /*name of another template in Databasename 1*/

$                           /* end of Databasename 1                     */
Databasename2               /*another database user allowed access to   */
Templatename3
.
.
.
$                           /*  marks end of database                    */
Databasename3
$
$                           /*  two '$' marks end of user profile        */
```

Figure 3.1 User Profile File Format

```
struct databases {
        char dbname[lngth+1];          /*database name in usr prfl   */
        struct templates *temps;       /*pnts to allowable templtes  */
        struct databases *nxtdbs;      /*pnts to next database in    */
                                       /*user profile                */
}

struct templates {
        char tempname[lngth+1];        /*template name in database   */
        struct attributes *attrs;      /*pnts attributes in database */
        struct templates *nxttemps;    /*pts to next template in db  */
}

struct attributes {
        char attrname[lngth+1];        /*attribute name in template  */
        char retr[lngth 1+1];          /* contains operation info.   */
        char insrt[lngth+1];           /* It can contain template,   */
        char updt[lngth+1];            /* security, or attribute     */
        char delt[lngth 1+1];          /* info.                      */
        struct attributes *nxtattrr;   /*next attribute in template  */
}
```

Figure 3.2 User Profile Linked-List Coding Structure

opportunity to check and modify this data prior to its being written to the user-profile file.

A similar method to creating a new user profile is used to modify an existing user profile. DBA specifies which user profile is to be modified. The file corresponding to the user name and user identification number is read into the user profile linked-list structure. Modifications are then easily made by replacing data values or by resetting pointers.

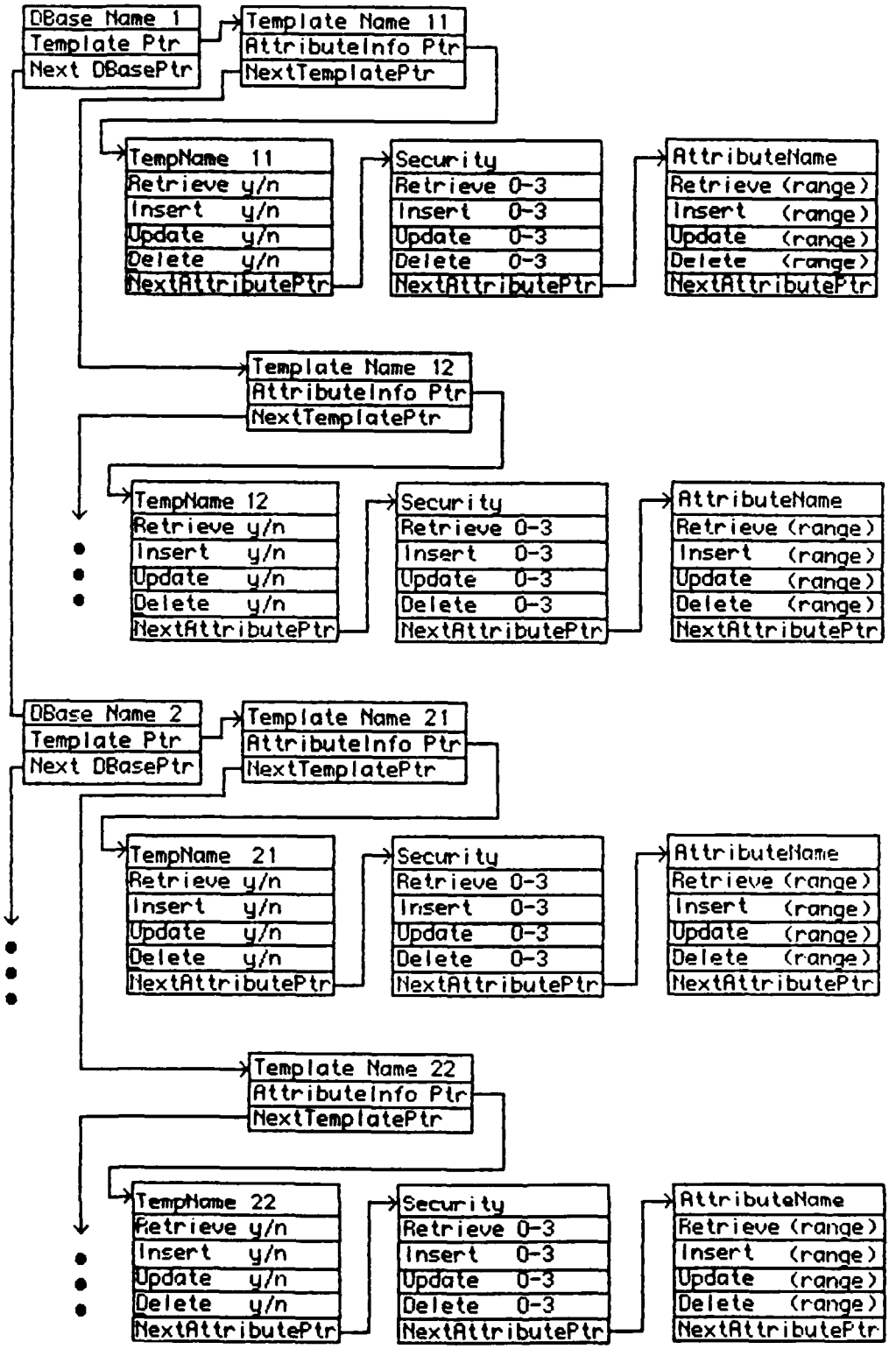

Figure 3.3 User Profile Linked-List Data Structure

The key point of this user profile mechanism is that only DBA and his/her trusted system processes can have access to the user-profile files. No user is able to modify his/her own, or any other user's profile.

B. QUERY MODIFICATION

1. The High-Level Design of Query Modification

The method to utilize the user-profile information is called Query Modification, or, simply, Query Mod. The methodology is straightforward. The Query-Mod routine will accept a query from a user request and modify the request based on that user's profile information. This method involves adding to the existing request attribute value pairs obtained from the user profile that belongs to the user making the request. Attribute-value pairs have been discussed in Chapter II. This can have the effect of further specifying the data to be accessed if the added attributes are key attributes. This is because the Query-Mod mechanism forces certain attribute-value pairs to be present in the (modified) request. These newly inserted attribute-value pairs force the request to be more specific as to the desired data. As a result, it will be likely that fewer clusters of records will have to retrieved. Therefore, Query Mod makes the execution of the request more efficient, i.e., increasing the access precision. This is because Query Mod is done prior to any actual data retrieval. As a result less unwanted data will be brought

into the memory that would need to be passed through once the data has been retrieved. The benefit of Query Mod with respect to security can clearly be seen here. Because Query Mod is done prior to data retrieval, data which the user is not authorized to access is never brought into the main memory. This is because due to Query Mod, the clusters containing unauthorized data with respect to the user are never considered for retrieval. It must be noted that this is true if all attributes used from the user profile are key attributes. Because the attributes in AT, DDIT, and CDT are exclusively key attributes, only those predicates in the user profile that contain key attributes will effect access precision. Predicates containing non-key key attributes in the user profile, while not improving access precision, do produce finer granularity in the end product for the user. It is the information contained in the user profile that determines what is authorized or unauthorized for that user.

2. The Detailed Design of Query Modification

A query is brought into the Query Mod mechanism in the form of a two dimensional array called a Request Table, or Reqtbl. The bounds of the array are specified at system boot-up by the specification of the maximum number of attributes and templates allowable per database. The query is parsed for syntactic correctness prior to being passed onto Query Mod.

Two primary data structures are utilized in Query Mod. The first is the user profile in its linked-list format in Figure 3.3. The second data structure is Reqtbl which as mentioned earlier is the two dimensional array that contains the query. Reqtbl format is illustrated in Figure 3.4.

Prior to entering the Query Mod mechanism, a user access to the system and to the particular database must be commenced. This is accomplished by first requiring the user to enter his/her name and password. These items are checked against the user's profile for access validation. If this access is valid, than the user's profile is read into the linked-list structure specified in Figure 3.3. The user is then required to enter the name of the database from which queries are made. This database name is checked against all the database names in the user profile (now in linked-list form). If a match is not found, the user is notified that he/she is not authorized to access this particular database. If a match is found, then pointers are set to mark the beginning and end of the user profile data that refers to this particular database. These pointers are used by both the query parser as well as the Query Mod mechanism. How the Query Mod mechanism utilizes this information is discussed in the following section.

```
Reqtbl[0]   /* beginning of query marker                              */
Reqtbl[1]   /* traffic ident. number                                  */
Reqtbl[2]   /* request number                                         */
Reqtbl[3]   /* route type                                             */
            /* above not necessary for Query Mod                      */
Reqtbl[4]   /* request type i.e. insert, delete etc.                  */
Reqtbl[5]   /* Number of predicates. predicate = attrib. name         */
             /* followed by relational operator followed by value     */
Reqtbl[6]   /* 'TEMP'                                                 */
Reqtbl[7]   /* '='                                                    */
Reqtbl[8]   /* Name of template with requested database               */
Reqtbl[9]   /* 'SECURITY'                                             */
Reqtbl[10]  /* relational operator, usually <=                        */
Reqtbl[11]  /* security value: 0-unclass,1-confid,2-secret            */
            /* 3-top secret                                           */
Reqtbl[12]  /* zero or more query predicates.                         */
            /* '$' marks end of predicates                            */
Reqtbl[13]  /* '*' marks beginning of target list. target             */
            /* list is where user states what values he/she           */
            /* wants to see as a result of the query                  */
Reqtbl[14]  /* tgt name = attribute name                              */
Reqtbl[15]  /* can have zero or more targets. list of target          */
            /*names terminated by a '*'                               */
Reqtbl[16]  /* sort field attribute. User can state if the             */
            /* response to the query is to be sorted based            */
            /* on attribute. must be one of the target list           */
            /* attribute names                                        */
Reqtbl[17]  /* ']' marks end of query                                 */
```

Figure 3.4 User Profile Request Table (Reqtbl) Format

3. An Algorithm for Query Modification of the Insert Operation

It is first determined what type of request the query is (i.e., insert, delete, retrieve, etc.). This is determined by examining Reqtbl[4]. An insert request is processed in a slightly different manner, because it does not require a relational operator for its predicates. The case of an insert request is examined in this section.

The format for an insert Reqtbl is illustrated in Figure 3.5. (Note that no relational operators are needed within the attribute-value pairs).

The Query Mod algorithm for an insert request is as follows:

1. For the first step of the insert, Query Mod is to verify that the attribute name, TEMP in Reqtbl (Reqtbl[7] has a matching attribute name, TEMP, in the user profile.

2. The next step is to verify that this user is authorized to perform an insert operation on this template, therefore, the file. This is done by checking the list of operations following the template name. A 'Y' or 'y' must exist next to the operation, insert, for this request to be valid.

3. The query is then checked to make sure that the security level has been included (i.e., in Reqtbl[8] and [9]). The security attribute is always included in both the request and the user profile if they have been generated by the database software. However, if they are created manually (i.e., via an editor), the user may have failed to include these attributes. If a security attribute does not exist in either the request or the user profile, the request is disallowed. This is because the security attribute value must be specified either by the user for the request or by DBA for the user profile. It is not possible to decide upon a default security value in either of these cases in order to allow the query to continue and still ensure adequate security. However,

```
Reqtbl[0] = [                    /* beginning of request                */
Reqtbl[1] = 0001                 /* traffic unit number                 */
Reqtbl[2] = 1                    /* request number                      */
Reqtbl[3] = 4                    /* route type                          */
Reqtbl[4] = 4                    /* request type; insert = 4            */
Reqtbl[5] = x                    /* number of attributes                */
Reqtbl[6] = TEMP                 /* required                            */
Reqtbl[7] = templatename         /* name of template within database    */
Reqtbl[8] = SECURITY             /* required                            */
Reqtbl[9] = x                    /* security level value 0,1,2,3        */
Reqtbl[10]= attributename1       /* attribute name within template      */
Reqtbl[11]= attributevalue
Reqtbl[12]= attributename2       /* attribute name within template      */
Reqtbl[13]= attributevalue
.
.
.
Reqtbl[x] = #                    /* end of attribute value pairs        */
Reqtbl[x+1] = ]                  /* end of request                      */
```

Figure 3.5 Insert Reqtbl Format

there is an exception in a retrieve-type request where the lowest security level is also the default security level for the user profile and request.

4. After the security attribute names have been verified, their values are checked to ensure that the security level in the request (i.e., Reqtbl[9]) is less than or equal to the security level in the user profile. If this comparison fails, the request is disallowed.

5. The final step is to check the remaining items in the request (i.e., Reqtbl[10]--End of Query). These are the attribute-value pairs to be inserted into the database. It must be verified that the values fall within the insert ranges specified in the user profile. This is effected by reading the information from the user-profile linked list that pertains to allowable insert ranges into a separate, working linked list. Each attribute name in the request is compared against this list to find a match on a user-profile-specified attribute name. If a match is

not found, the request is discontinued because the attribute name in the request does not exist in the database's template. Once a match is found, it is verified that the request values fall within the range specified in the user profile for that attribute. If a requested insert value falls outside a specified range, the request is discontinued.

As can be seen, the Query Mod mechanism for an insert type request does not necessitate actual modification to the query. The Query Mod mechanism simply acts as a 'security parser' by verifying that the requested operations and the values to be inserted are authorized. No modification to the request is necessary because as stated earlier in step three, no default value is utilized. Instead, proper values are set in the user profile that will ensure proper security and data integrity. However, the other types of requests: retrieve, update, delete, and retrieve-common, can actually be modified or appended to ensure their compliance to the security requirements specified in the user profile.

4. An Algorithm for Query Modification of Non-Insert Operations

a. A General Discussion

As discussed earlier, the modification algorithm for an insert-type request differs from the non-insert-type requests. The difference between the two can be shown by comparing the respective Reqtbl for inert and non-insert types. The non-insert type of requests requires relational operators in their predicates whereas the insert type of requests does not. The algorithm for the insert request

simply verified that the insert operation is allowed for the requested attributes and that their corresponding values are within the allowable ranges permitted by the user profile. The algorithm for the non-insert request needs more extensive processing and verification; however, the data structures (i.e., Reqtbl and the user profile) remain essentially the same except where explicitly shown.

Unlike the Query Mod algorithm for the insert type of requests, the algorithm for a non-insert request will involve actual modification to the query. The modified query is the result of predicates being appended to the request. The non-insert requests are retrieve, update, delete, and retrieve-common. These newly appended predicates come from the user profile. The result of this modification is that the request may contain two predicates that refer to the same attribute. For example:

```
PartNumber <= 2000  /*original request from the user*/

PartNumber <= 1000  /*predicate from the user-profile*/
```

In this example the user requested to see a PartNumber less than or equal to 2000. The query is appended with the user-profile predicate for this attribute, PartNumber <= 1000, meaning that DBA has specified that this

user is authorized to access only PartNumbers less than or equal to 1000. It can be seen that the intersection of these two predicates results in an authorized level of request/access, i.e., PartNumber <= 1000. This intersection process does not take place in the Query Mod mechanism, but is handled later just before the access takes place. In other words, the Query Mod mechanism only modifies a user request by appending predicates. The modified request which effects the intersection process will be executed by our new access control mechanism (which is discussed in the sequel).

b. A Detailed Discussion

The Query Mod algorithm for a non-insert type of requests is as follows:

1. This step is similar to the first step of the Query Mod algorithm for an insert request; the template name in the request (Reqtbl[8]) is compared to the user profile's template (file) name.

2. The user profile is then checked to see if the desired operation type (retrieve, retrieve-common, update, delete) is authorized for the template.

3. The next step is to verify that the security attribute has been included in the request (Reqtbl[9],[10], [11]), and that it is also in the user profile. If the security attribute is not found in the request, the user profile security level is inserted into the request. If the security attribute is not found in the user profile, insert the lowest level of security (0) into the request.

4. The next step is to verify that the requested security level is within the security level specified in the user profile. The user profile allows DBA to specify different accesses for different security levels based on the request type (insert, delete, etc.). If the requested security level is out of range, the security level in the request (Reqtbl[12]) is overwritten with the security level specified in the user profile.

5. The final step is to modify the query predicate list with predicates from the user profile. As shown in Figure 3.1, the user profile contains the access control information for each attribute of a given database. This access control information is in the form of predicates and are related to each attribute based on operation, i.e., retrieve, insert, etc. The operation selected is based on the query. These predicates, which articulate access control for a given operation, are appended onto the predicate list.

The result is a query now modified with the desired access control. Every user query will be modified by this method, which is totally transparent to the user. As a result of this modification the predicate list has more predicates which therefore better articulate or define the clusters that will answer a query. Because the modified query is better able to define the desired clusters, fewer clusters and therefore fewer records are retrieved. The savings, as a result, are significant because of the inherent slowness of data retrieval from the secondary storage.

C. AN EXAMPLE OF QUERY MODIFICATION

The following example is provided to further illustrate the Query Mod mechanism.

1. A Sample User Profile

Figure 3.6 is provided in support of the example. Reading the example in Figure 3.6 from the top to the bottom, we note that this user's name is Smith, his/her ID number is 9999, and his/her password is secretpassword. Smith has accesses to only one database called SCHOOL. To

```
Smith
9999
secretpassword
SCHOOL
Course
RETRIEVE y
INSERT y
UPDATE y
DELETE
SECURITY
RETRIEVE 2
INSERT 1
UPDATE 1
DELETE 0
CRSNUM
RETRIEVE CRSNUM 2 CS999 CRSNUM 4 CS000
INSERT CRSNUM 2 CS999 CRSNUM 4 CS000
UPDATE CRSNUM 2 CS999 CRSNUM 4 CS000
DELETE
ROOM
RETRIEVE ROOM 4 000
INSERT ROOM 4 000
UPDATE ROOM 4 000
DELETE
@
Student
RETRIEVE Y
INSERT
UPDATE
DELETE
SECURITY
RETRIEVE 0
INSERT
UPDATE
DELETE
STUDNAME
RETRIEVE STUDNAME 4 a
INSERT
UPDATE
DELETE
GPA
RETRIEVE GPA 4 0
INSERT
UPDATE
DELETE
$
$
```

Figure 3.6 Sample User Profile

help describe this user's profile, some of the authorized options for Smith based on this profile are provided. For example, there are two templates (therefore, two files) in the database SCHOOL to which Smith can access: Course and Student. For the records of Course, Smith is authorized with retrievals, insertions, updates, but not deletions. Smith is restricted to retrieving only up to secret information; and inserting and updating confidential information. Further, restrictions are dictated regarding the attributes in template Course. For example, Smith has the access to records of Computer Sciences classes CS000 to CS999 but no others. Here, numbers are used to represent relational operators: 1 <; 2 <=; 3 >; 4 >=; 5 =; 6 !=. Therefore, Smith is authorized to retrieve those values greater than or equal to zero for the attribute GPA that are unclassified. The attribute GPA is in the template Student which is in the database SCHOOL.

2. An Example of a Request

The following is an example query that is originated by the user Smith whose profile has been provided in the previous figure.

```
[RETRIEVE((TEMP=Course) and (SECURITY<=3)and(CRSNUM<=
CS1200))(CRSNUM,ROOM)]
```

This request is of the retrieve type, the retrieval will be made from template Course; and the security level of the information requested is less than or equal to top-secret. The actual query is for all Course information that is top-secret or below with course numbers (CRSNUM) less than or equal to CS1200. The information desired is the CRSNUM and ROOM values that satisfy this request.

Comparison of Smith's request against Smith's profile cause several items to note. First, Smith is authorized to retrieve secret or below information, the query is requesting top-secret and below. Thus, the request is out of range of the security attribute values. Also note that Smith is requesting to see all CRSNUM's and ROOM's that meet the requirements of the predicates; however, Smith is limited to see only Computer Science (CS) courses from CS000 to CS999. The following section demonstrates how the Query Mod mechanism handles these restrictions. Figure 3.7 illustrates how ABDL query appears as in its Reqtbl form.

3. The Query Modification Algorithm

At this point in the algorithm, it assumed that Smith has already logged on, and requested to use the database, School. This means that the access to this database has been verified, and that the user profile has been read into its linked-list form.

The next step in the algorithm is then to determine whether the requested operations on the template, Course

```
Reqtbl[0] = [
Reqtbl[1] = 9999          /* traffic ID number                    */
Reqtbl[2] = 1             /* request number                       */
Reqtbl[3] = 1             /* route type                           */
Reqtbl[4] = 4             /* request type retrieve = 4            */
Reqtbl[5] = 3             /* number of predicates                 */
Reqtbl[6] = TEMP          /* required                             */
Reqtbl[7] = 5             /* relational operator 5 = '='          */
Reqtbl[8] = Course        /* template name                        */
Reqtbl[9] = SECURITY      /* required                             */
Reqtbl[10]= 2             /* relational operator 2 = '<='         */
Reqtbl[11]= 3             /* security level 3 = top secret        */
Reqtbl[12]= CRSNUM        /* predicate attribute name             */
Reqtbl[13]= 2             /* relational operator 2 = '<='         */
Reqtbl[14]= CS1200        /* predicate value                      */
Reqtbl[15]= $             /* marks end of predicates              */
Reqtbl[16]= *             /* beginning of target list             */
Reqtbl[17]= CRSNUM        /* attribute name                       */
Reqtbl[18]= ROOM          /* attribute name                       */
Reqtbl[19]= *             /* end of target list                   */
Reqtbl[20]= ]             /* marks end of request                 */
```

Figure 3.7 ABDL Query in Reqtbl Form

(Reqtbl[8]), are authorized. This is done by finding the matching template name in the user profile. Because this is a retrieve-type request, permissions for retrieve operations on this template must be granted in the user profile. In this case, the retrieve operation is authorized.

The next step is to determine that security attributes are indeed included in both the user query and the user profile. Both of these do exist in our example, and it is determined that Smith is authorized to retrieve up

to and including secret information. However, the query is requesting information less than or equal to top-secret. Query Mod handles this by replacing the security value in the request (Reqtbl[11]), i.e., 3 for top-secret, with the security value in the user profile, i.e., 2 for secret.

At this point in the algorithm the predicate list has been reached. Therefore, all of the predicates in the user profile for each attribute that relate to the retrieve operation are appended onto the predicate list. Specifically, the template upon which the query is being made has two attributes, CRSNUM and ROOM. Because this is a retrieve-type request, only the predicates that relate to the retrieve operation are selected. The predicates selected for CRSNUM are CRSNUM 2 CS999 and CRSNUM 4 CS000. The predicate selected for ROOM is ROOM 4 000. These three predicates are appended onto the predicate list. The result is a different query than whathas been started with. This newly modified query is shown in Figure 3.8. As can be seen in Figure 3.8, the query has been updated with a new security value, and with three new predicates as a result of Query Mod. This new query can now retrieve data that is authorized to be viewed by Smith.

```
Reqtbl[0] = [
Reqtbl[1] = 9999
Reqtbl[2] = 1
Reqtbl[3] = 1
Reqtbl[4] = 4
Reqtbl[5] = 5
Reqtbl[6] = TEMP
Reqtbl[7] = 5
Reqtbl[8] = Course
Reqtbl[9] = SECURITY
Reqtbl[10]= 2
Reqtbl[11]= 2                    /*updated security level value  */
Reqtbl[12]= CRSNUM               /*original predicate list item  */
Reqtbl[13]= 2
Reqtbl[14]= CS1200
Reqtbl[12]= CRSNUM               /*inserted from the user profile*/
Reqtbl[13]= 2
Reqtbl[14]= CS999
Reqtbl[15]= CRSNUM               /*inserted from the user profile*/
Reqtbl[16]= 2
Reqtbl[17]= CS999
Reqtbl[18]= ROOM                 /*inserted from the user profile*/
Reqtbl[19]= 4
Reqtbl[20]= 000
Reqtbl[21]= *
Reqtbl[22]= CRSNUM
Reqtbl[23]= ROOM
Reqtbl[24]= *
Reqtbl[25]= ]
```

Figure 3.8 Modified ABDL Query in Reqtbl Form

IV. MULTILEVEL SECURITY

A. INTRODUCTION

Chapter II introduced the notion of attribute value pairs, and how these are used via three meta data tables (AT,DDIT,CDT) clusters are formed. Because of the nature of these clusters an access precision of one was achieved. Chapter III used the idea of attribute value pairs to show how access control could be specified down to the attribute level for each user. Chapter III also introduced a security attribute value pair, thereby introducing the notion of multilevel security. Multilevel security is concerned with the application and implementation of providing access control based on these security levels. This chapter will discuss how multilevel security, using this security attribute, is implemented into the database system as described in Chapters II and III.

B. HIGH LEVEL DESCRIPTION OF MULTILEVEL SECURITY

1. High Level Description

As stated in the previous section, Chapter II introduced three meta data tables, AT, DDIT, and CDT. Multilevel security, as suggested by Hoppenstand, is implemented by using these tables, and using security as an attribute value pair, with the attribute name being "SECURITY," and the corresponding values being security

levels as specified by application. This security attribute however is not generally treated the same as other attributes. These differences will shown in the remainder of this chapter.

The first difference in the way the security attribute is handled is that it is not introduced in either AT or DDIT. The reason for this is that references to security attributes should be kept to a minimum. Using this reasoning, there is no need to introduce the security attribute until CDT. It is in this table that the security attribute is first used. The following Figure 4.1 is based on Figure 2.1. Figure 4.1 shows the result of introducing multilevel security. More specifically it shows the introduction of security attribute value pairs. For this example, four levels of security are presupposed: Unclassified (U), Confidential (C), Secret (S), and Top Secret (TS).

Notice that Figure 4.1 is partitioned into four parts, more specifically four CDT's, based on security levels. The reason for this will be discussed. Notice that Figure 4.1 shows that four clusters, C1, C6, C11 and C16, are defined by the same Descriptor Id Sets, {D11,D21,D31}, if the security attributes for each are excluded. Therefore, the security attributes are necessary to appropriately cluster the corresponding records. As suggested in Hoppenstand, all records must be defined in terms of a security attribute. To illustrate this, consider the

Id	Desc-Id Set	Record-Id
C1	{D11,D21,D31,TS}	R1
C2	{D12,D21,D31,TS}	R7
C3	{D14,D21,D32,TS}	R18
C4	{D14,D23,D32,TS}	R27
C5	{D13,D23,D32,TS}	R24

Id	Desc-Id Set	Record-Id
C6	{D11,D21,D31,S}	R2
C7	{D12,D21,D32,S}	R8,R9
C8	{D13,D22,D31,S}	R12
C9	{D13,D23,D32,S}	R25
C10	{D14,D23,D32,S}	R28

Id	Desc-Id Set	Record-Id
C11	{D11,D21,D31,C}	R3
C12	{D11,D21,D32,C}	R5
C13	{D12,D21,D32,C}	R10,R11
C14	{D13,D22,D31,C}	R13
C15	{D14,D21,D32,C}	R19

Id	Desc-Id Set	Record-Id
C16	{D11,D21,D31,U}	R4
C17	{D11,D21,D32,U}	R6
C18	{D13,D22,D31,U}	R14
C19	{D14,D21,D31,U}	R17
C20	{D14,D21,D32,U}	R20,R21,R22
C21	{D13,D23,D31,U}	R23
C22	{D14,D23,D32,U}	R26

Figure 4.1 Multiple CDTs as a Result of Multilevel Security Introduction

following example. Suppose a record was allowed to be inserted without a security attribute. The result would be similar masking off the security attribute mentioned previously: the record could potentially be defined over four different security levels, and therefore could be defined as being simultaneously contained in four clusters. As a result, the clusters would no longer define disjoint sets of records, and therefore not support the mathematical notion of equivalence classes upon which ABDL is based. Therefore each record must be defined in terms of a security attribute.

2. Multiple CDTs

It has been shown to this point that the security attribute must be included in all cluster definitions. If four security levels were used, i.e., Unclassified, Confidential, Secret, and Top Secret, the result would be to potentially increase the size of the Cluster Definition Table (CDT) by a factor of four. This is because each cluster could be further defined by each classification level, in this case, four levels. This increase in size is a worst case scenario. If no records of a given classification exist for a particular cluster definition, then no cluster definition will be formed for it. Regardless of this particular occurrence, the size of CDT is greatly increased. The characteristics of this new form of CDT presents two problems.

The first problem is simply that because of its increased size, the time necessary to find the appropriate clusters based on Descriptor-Id Sets is increased as well. The second problem is that the Pass Through problem, discussed in Chapter I, has only been one step removed. This is because even though the potential of searching through base data for which a user may not be cleared for has been avoided, the meta data, i.e., CDT, which refers to data for which a user may not be cleared for, is brought into memory and searched. Though this does not cause a pass through problem, a more secure method that would only search those CDTs that refers to data cleared for a user would be preferable.

A solution to both of the above posed problems is to partition this single large CDT into separate CDTs based on security level. The result is Figure 4.1, which demonstrates distinct CDTs that contain the meta data that refers to data of only one security level.

The result of these multiple CDTs is twofold. The first is that the amount of meta data that needs to be searched is potentially reduced by a factor of the number of security levels. This is a potential savings that depends on the query and security level. If a query requests only data of a particular security level, then the maximum gain in efficiency is realized because only one CDT will be searched as a result. However, a more likely query will

request data that will cover a range of security levels. This is due to the fact that a user is authorized access to data at his/her security level and below. Therefore, the amount of savings from multilevel security is determined by the security level itself. For example, if the security level is the highest level (i.e., Top Secret), then the savings is minimal because all CDTs will need to be searched. However, if the security level is the lowest level, then the maximum savings in terms of efficiency is obtained because only one CDT will be searched. The average savings over time would be a 50% increase in efficiency. This is so because assuming that data of each security level has equal likelihood of being requested, then on the average only half of CDTs will be searched as a result.

The second result of creating multiple CDTs based on security level is a more secure solution to the pass-through problem. The pass-through problem is solved in a more secure fashion because as a result of multiple CDTs, not only is the base data not authorized for a user not searched, the meta data that references this unauthorized base data is also not searched. Multiple CDTs makes this possible because only those CDTs with the security level authorized for a given request are actually searched in memory.

C. A DETAILED DESCRIPTION OF MULTIPLE CDTS

This section will describe in greater detail how multiple CDTs are utilized in response to a query. This description will follow a query after it has been through the Query Mod process.

In Chapter III the notion of a request table (Reqtbl) was introduced, and was defined as an array that largely contained attribute value pairs that constituted a query. This request table went through the Query Mod process described in Chapter III. AT and DDIT tables are then utilized to construct Descriptor-Id Sets, introduced in Chapter II, which in turn are used to find the clusters and ultimately to access record addresses of base data that responds to the query. This process will not be discussed in detail, but will only be explained as far as necessary to explain the lower- level details of multilevel security.

The request table is passed to each of the backends in a data structure called a traffic unit. The name of the database on which the queries will be made is included in this traffic unit. The name of the database is provided by the user, and it is verified that the user has access to this database during Query Mod. When each of the backends receive this traffic unit, the database name is used to initiate the construction in the memory of the meta data that will describe this database names base data. What is

constructed is a detailed description of AT, DDIT and multiple CDT tables.

The implementation enforces multilevel security utilizing multiple CDTs in the following fashion. Figure 4.2 shows the general notion of the implementation.

Each rectangle in Figure 4.2 represents CDT, and each is identified with its respective security level. As shown, there exists individual pointers for each security level. In this example, there are four pointers, one for each of Unclassified, Confidential, Secret, and Top Secret. Only one of these pointers will be referenced in response to a query. The security clearance level of the user, or the security level specified in the query, is the deciding factor for which pointer is to be utilized. As stated in Chapter III, if the user does not specify a security level in his/her query, the default is the lowest security clearance level. If the user included a security level in his/her query, and that specified security level is less than or equal to that users security clearance level, then it is that security level which decides which pointer is utilized.

Figure 4.3 shows the simplified data structure of a traffic unit with out multilevel security. The traffic unit contains a pointer to meta data (CDT) that describes the database upon which queries will be made. Also included in

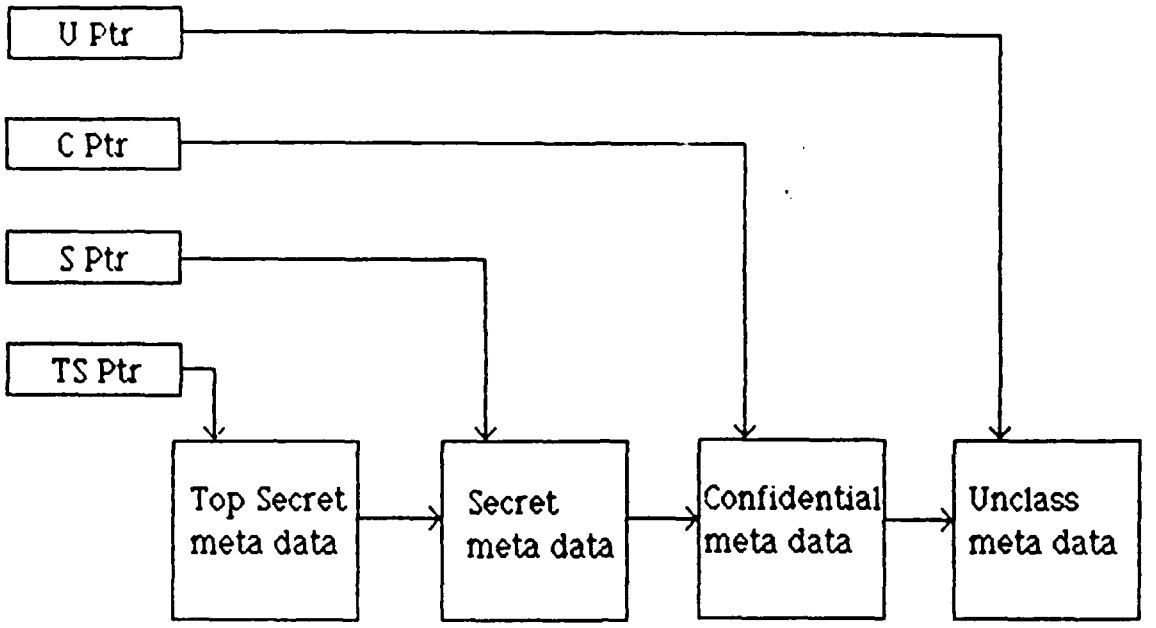

Figure 4.2 High Level Multilevel Security Implementation

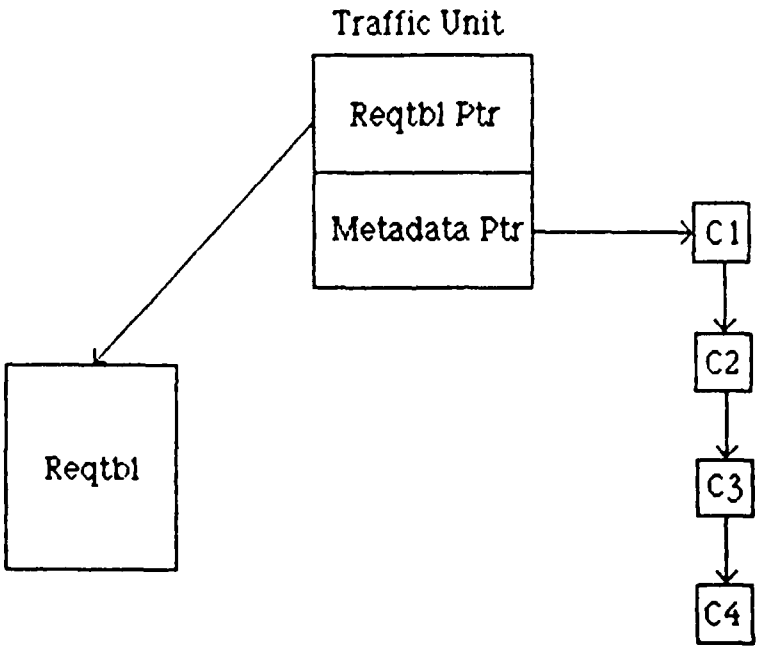

Figure 4.3 Traffic Unit Data Structure

the traffic unit is the request table (reqtbl) which contains the query itself.

The meta data in this traffic unit consists of a singly linked list of clusters that constitute CDT. When a search of the clusters is made in response to the query, the list is entered at the first cluster, C1. The search then continues down th list following the pointers from cluster to cluster, until the end of the list is encountered.

Multilevel security is implemented by modifying the meta data pointer. Figure 4.4 shows a lower level view of the result of this modification. The single meta data pointer is replaced by an array of pointers. This number of pointers in the array is equal to the number of security levels in the database. In this example, there are four security levels. Therefore, each pointer in the array represents a security level.

Notice how Figure 4.4 models Figure 4.2. The linked list of clusters is entered based on security level. Because the list is singly linked, a CDT search entering at a given security level can not move up the list. That is, a cluster search is forced to search at the security level at which it was entered, and those clusters of lower security levels. For example, a query entering with a Confidential security level can only search clusters at the Confidential and Unclassified levels. The clusters at the Top Secret and Secret security levels will never be referenced. If

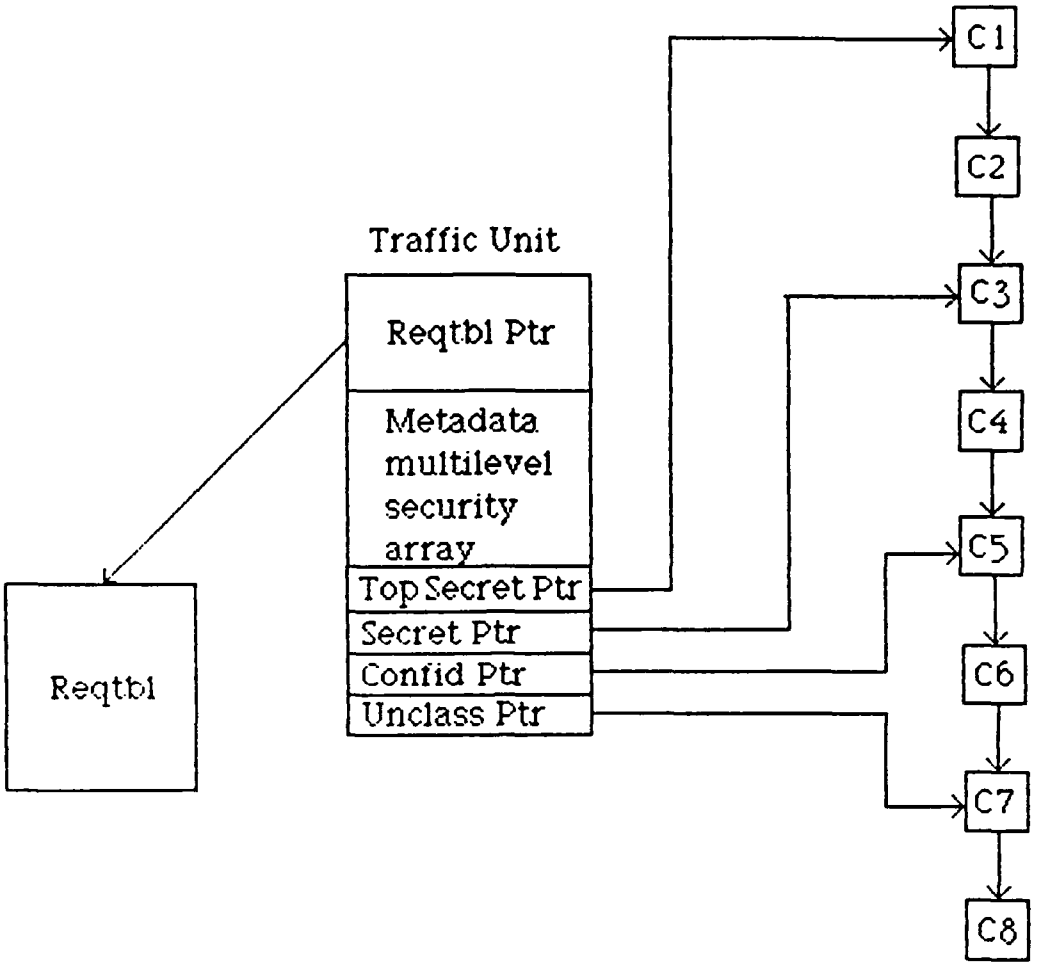

Figure 4.4 Multilevel Security Traffic Unit Data Structure

clusters at the Confidential and Unclassified security levels are found in response to the query, the record addresses corresponding to these clusters are returned, and these records will ultimately be retrieved. Note that this implementation directly supports the notion of multilevel security in that access is granted only at the specified security level and below.

V. CONCLUSIONS

A. SUMMARY OF CONTRIBUTION

This thesis has shown for the first time, a frontend (Query Mod) and backend (Multilevel Security) approach to database security, implemented into a single database system. Hoppenstand's proposal on the structures of multilevel secure databases was incorporated into the experimental database management system, the Multi-Backend, Multi-Lingual, Multi-Model Database System. An access control mechanism for DBMS, derived from the Query Modification research of D.K. Hsiao, M. Stonebraker, and E. Wong was designed and implemented to regulate and verify access to the multilevel secure database generated from Hoppenstand's proposal.

The frontend (Query Mod) mechanism provides a high degree of access control down to the attribute level. The backend (Multilevel Security) mechanism provides for data access in such a way as to eliminate the pass-through problem. Both of these mechanisms function together to provide an absolute access precision. The result is a highly secure and highly efficient database system. Security mechanisms on contemporary database machines typically inhibit system performance. The Query Mod and Multilevel Security mechanisms, however, enhance system

performance while simultaneously providing a higher degree of security.

B. REMAINING ISSUES AND FUTURE WORK

Two areas of work may follow: (1) a formal proof that Hoppenstand's proposal is secure and then a formal proof that the implementation of Hoppenstand's proposal is secure; and (2) benchmarking of system efficiency after multilevel security and access control implementation for comparison with [Ref. 9] to determine the degree of system efficiency improvement. Because there is an infinite number of possible database security test runs, a formal proof is the only way to completely prove a secure system and therefore formally show the elegance of the multilevel security proposal. The benchmarking of the system will serve to experimentally solidify the expected system performance enhancement due to Multilevel Security and Query Modification.

LIST OF REFERENCES

1. Hoppenstand, G.S., Secure Access Control with High Precision, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

2. Hoppenstand, G.S. and Hsiao, D.K., "Secure Access Control with High Access Precision: An Efficient Approach to Multilevel Security," in: Database Security, II--Status and Prospects, C.E. Landwehr, Ed., North-Holland, pp. 167-176.

3. Kerr, D.S., Hsiao, D.K., and Madnick, S.E., Computer Security, Academic Press, 1979.

4. "Department of Defense Trusted Computed System Evaluation Criteria," Department of Defense Computer Security Center, CSC-STD-001-83, 15 August 1983.

5. Hsiao, D.K., "Access Control in an On-line File System," File Organization--Selected Papers from FILE 68, An IAG Conference, Student Literature Ab, Lund, Sweden, November 1968.

6. Stonebraker, M. and Wond, E., "Access Control in Relational Database Management System by Query Modification," Proceedings of 1974 ACM National Conference, ACM Press, 1974.

7. Hsiao, D.K. and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970.

8. Hsiao, D.K. and Kamel, M.N., "Heterogeneous Databases: Proliferations, Issues, and Solutions," IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 1, March 1989.

9. Hall, James E., Performance Evaluation of a Parallel and Expandable Database Computer--The Multi-Backend Database Computer, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Commandant of the Marine Corps Code TE06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
5.	Commandant of the Marine Corps C21-0 Attn: Maj. L. Kratochvil, USMC Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
6.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
7.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
8.	Professor David K. Hsiao, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	4
9.	Lieutenant Matthew J. Kohler, USN 341 Shenley Drive Erie, Pennsylvania 16505	2

10. Captain Shawn W. Stroud, USMC
4405 Fegenbush Lane
Louisville, Kentucky 40218 — 2

11. Debbie Gaiser
2739 Eaton Street
New Orleans, Louisiana 70131 — 1

12. Marciano P. Pitargue
6607 Kaiser Drive
Fremont, California 94555 — 1